Compositional Invariant Generation via Linear Recurrence Analysis

Azadeh Farzan and Zachary Kincaid

University of Toronto

Abstract. This paper presents a new method for automatically generating numerical invariants for imperative programs. Given a program, our procedure computes a binary input/output relation on program states which over-approximates the behaviour of the program. It is compositional in the sense that it operates by decomposing the program into parts, computing an abstract meaning of each part, and then composing the meanings. Our method for approximating loop behaviour is based on first approximating the meaning of the loop body, extracting recurrence relations from that approximation, and then using the closed forms to approximate the loop. Our experiments demonstrate that on verification tasks, our method is competitive with leading invariant generation and verification tools.

1 Introduction

Compositional program analyses operate by decomposing a program into parts, computing an abstract meaning of each part, and then composing the meanings. Compositional analyses have a number of desirable properties, including scalability, parallelizability, and applicability to incomplete programs. However, compositionality comes with a price: since each program fragment is analyzed independently of its context, the analysis cannot benefit from contextual information. This paper presents a compositional method for numerical invariant generation which, despite loss of contextual information, compares favourably with leading (non-compositional) verification techniques.

The analysis proposed in this paper aims to compute a *transition relation* which over-approximates the behaviour of a given program. The use of transition relations in compositional analysis (e.g., [23,25,1,21,17,5]) stems from the fact that they can be composed: for example, consider a program $P = P_1; P_2$ which consists of two sub-programs P_1 and P_2 which are executed in sequence. A transition invariant $[\![P]\!]$ for P can be computed by computing transition invariants $[\![P_1]\!]$ and $[\![P_2]\!]$ for the subprograms and then taking $[\![P]\!]$ to be the relational composition: $[\![P]\!] = \{(s, s'') : \exists s'.(s, s') \in [\![P_1]\!] \wedge (s', s'') \in [\![P_2]\!]\}$.

A crucial question is how to compute abstractions of loops (i.e., *loop summaries* [17]). Our analysis is based on a classical idea: find recurrence relations for variables modified in the body of a loop, and then use the closed forms for these recurrences as the abstraction of the loop. The focus of research on recurrence analysis has mainly been on computing the *exact* behaviour of a (necessarily)

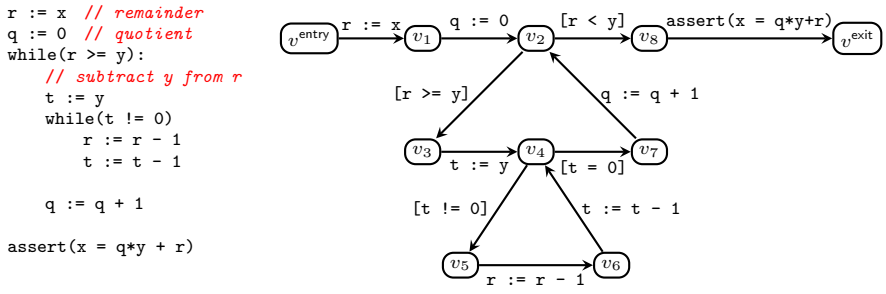
limited class loops, e.g. loops where the body is a sequence of affine assignments (see Section 6 for a discussion of related literature). We shift the goal to computing *over-approximate* behaviour of *arbitrary* loops. The main novelty of our approach is to make synergistic use of recurrence analysis and compositionality: on one hand, recurrence analysis can be used to compute accurate transition formulas for loops; on the other hand, transition formulas for loop *bodies* can be mined for recurrence relations to enable recurrence analysis.

Compositionality enables using recurrence analysis for arbitrary loops in two ways. First, the fact that the transition formula for a loop is computed from a transition formula for its body makes the control structure of the loop irrelevant (e.g., whether it is a sequence of assignments or contains branching or nested loops – its transition formula is just a formula). Second, having access to a loop body formula when computing a loop summary opens the door to using Satisfiability Modulo Theories (SMT) solvers to extract a broad range *semantic* recurrences. In particular, our analysis is able to exploit *approximate recurrences* (inequations over linear terms) to compute interesting loop invariants even for variables which do not satisfy recurrence equations in the classical sense, thus extending the applicability of recurrence-based invariant generation and overcoming a major barrier in its practical use.

In summary, this paper presents a compositional method for generating numerical invariants (polynomial inequalities of unbounded degree among integer and rational variables) for programs. The main technical contributions are as follows.

1. We give a method for computing abstractions of loops using summaries for their bodies. This allows our analysis to apply to arbitrary code (with nested loops, unstructured loops, and arbitrary branching). It also makes it possible to use SMT solvers to extract *semantic* recurrence relations rather than syntactic recurrences obtained by pattern-matching source code.
2. We identify a class recurrence (in)equations that can be efficiently extracted from loop bodies using SMT solving technology and solved using simple linear algebra.
3. We give a *linearization* algorithm which enables tractable (but necessarily approximate) reasoning about non-linear formulas over rationals and integers (Section 4).
4. We collect ideas from a diverse range of sources (including algebraic program analysis [10], recurrence analysis [1,15,2], linearization [20], and symbolic abstraction [26,22,19]), and synthesize them into a cohesive presentation which can be used as a foundation for futher research on recurrence analysis.

We implemented linear recurrence analysis and used it to verify assertions for a suite of benchmarks. Linear recurrence analysis is able to prove the correctness of more benchmarks in this suite than any of the leading verification tools for integer programs.



(a) Program text (b) Flow graph

Fig. 1. An integer division program, computing a quotient and remainder. Statements of the form $[\psi]$ represent *assumptions*; i.e., statements which block if ψ does not hold.

2 Overview

We will adopt a simple intraprocedural model in which a program is represented by a control flow automaton (CFA) where edges are labeled by program statements. Figure 1 depicts such a CFA for a program which computes the quotient and remainder of division of a variable `x` by a variable `y`. We use this model for the sake of simplicity and to help keep the presentation of our analysis short and self-contained. We hope that the basic idea behind the extension to procedures (implemented in the tool), using the analysis to compute procedure summaries [29], is clear without formal explanation.

Our analysis, linear recurrence analysis (LRA), is presented in the algebraic framework described in [10]. Suppose that we wish to prove that the assertion `assert(x = q*y + r)` always succeeds. We begin by computing the set of paths from v^{entry} to v_8 (the location corresponding to the `assert` statement in the CFA). This set of paths is represented by a *path expression* for the vertex v_8, which is a regular expression over an alphabet of control flow edges. In principle, this can be accomplished by Kleene's well-known algorithm for converting a finite automaton into a regular expression [14] (but more efficient algorithms exist [30]). For example, the following is a path expression for v_8:

$$\langle v^{\mathsf{entry}}, v_1\rangle \cdot \langle v_1, v_2\rangle \cdot \underbrace{\Big(\langle v_2, v_3\rangle \cdot \langle v_3, v_4\rangle \cdot \overbrace{\big(\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle \cdot \langle v_6, v_4\rangle\big)^*}^{\text{Inner loop}} \cdot \langle v_4, v_7\rangle \cdot \langle v_7, v_2\rangle\Big)^*}_{\text{Outer loop}} \cdot \langle v_2, v_8\rangle$$

Once we have a path expression representing the paths to v_8, we compute an over-approximation of the executions to v_8 by *evaluating* the path expression in some abstract domain. The main benefit of this algebraic framework is that an analysis is defined simply by providing an interpretation for each of the regular expression operators (sequencing, choice, and iteration, corresponding to the control structures of structured programs), and then we may rely on a path expression algorithm ([14,30]) to efficiently "lift" the analysis to programs with arbitrary control flow.

Formally, a program analysis (in the framework of [10]) is defined by an *interpretation*, which consists of a *semantic algebra* and a *semantic function*. A semantic algebra consists of a *universe* which defines the space of possible program meanings, and *sequencing*, *choice*, and *iteration* operators, which define how to compose program meanings. A semantic function is a mapping from control flow edges to elements of the universe which defines the meaning of each control flow edge. A path expression is *evaluated* by interpreting the individual edges using the semantic function, and interpreting the regular expression operators using the corresponding operators of the semantic algebra (to compose the interpretations of individual edges into interpretations of sets of program paths).

Keeping this overall algorithm in mind, we proceed to describe the interpretation which defines linear recurrence analysis.

LRA Universe. The semantic universe of LRA (i.e., the space of program meanings) is the set of (not necessarily linear) arithmetic *transition formulas*. If we let Var denote the set of program variables and Var' the set of "primed" copies of program variables, then a transition formula is an arithmetic formula with free variables in $\mathsf{Var} \cup \mathsf{Var}'$. Such a formula represents an input/output relation between program states.

LRA Semantic Function. The semantic function $[\![\cdot]\!]$ is a function that maps each edge of a control flow automaton to its interpretation as a transition formula. For example (again, considering Figure 1), we have

$$[\![\langle v^{\mathsf{entry}}, v_1\rangle]\!] = \boxed{r' = x \wedge \mathit{stable}(\{q,t,x,y\})}$$
$$[\![\langle v_1, v_2\rangle]\!] = \boxed{q' = 0 \wedge \mathit{stable}(\{r,t,x,y\})}$$
$$[\![\langle v_2, v_3\rangle]\!] = \boxed{r > y \wedge \mathit{stable}(\{q,r,t,x,y\})}$$

where for $X \subseteq \mathsf{Var}$, we have $\mathit{stable}(X) \triangleq \boxed{\bigwedge_{x \in X} x' = x}$; we use this to factor out equalities from the formulas and make them more legible. Boxes around formulas have no meaning, and are used only to make it easier to distinguish between equalities in formulas and the meta-language.

LRA Operators. The sequencing and choice operators of our analysis are defined as follows:

$$\varphi \odot \psi = \exists x''.\varphi[x''/x'] \wedge \psi[x''/x] \qquad \text{Sequencing}$$
$$\varphi \oplus \psi = \varphi \vee \psi \qquad \text{Choice}$$

(where $\varphi[x''/x']$ denotes φ with each primed variable x' replaced by its double-primed counterpart x'', and $\psi[x''/x]$ similarly replaces unprimed variables with double-primed variables).

The semantic function, sequencing, and choice operators are sufficient to analyze loop-free code. For example, we may consider how LRA computes a transition invariant for the body of the inner loop of Figure 1:

$$[\![\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle]\!] = [\![\langle v_4, v_5\rangle]\!] \odot [\![\langle v_5, v_6\rangle]\!]$$
$$= \boxed{t > 0 \wedge r' = r - 1 \wedge \mathit{stable}(\{q,t,x,y\})}$$

$$[\![\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle \cdot \langle v_6, v_4\rangle]\!] = [\![\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle]\!] \odot [\![\langle v_6, v_4\rangle]\!]$$
$$= \boxed{t > 0 \wedge r' = r - 1 \wedge t' = t - 1 \wedge \mathit{stable}(\{q, x, y\})}$$

The final step in describing our analysis is to provide a definition of the iteration operator ($\circledast$) of LRA. The idea behind the definition of the iteration operator is to use an SMT solver to extract recurrence relations from the loop body, and then use the closed form of these recurrences for the abstraction of the loop. We explain this in detail in Section 3. Here, we illustrate how LRA works on the running example to provide some intuition on the analysis.

After computing a formula φ_{inner} representing the body of the inner loop (as given above), we apply the iteration operator $\circledast$ to compute a formula representing any number of executions of the inner loop. The iteration operator begins by extracting the recurrence equations shown to the right. It then computes closed forms for these recurrences, also shown to the right (where $x^{(k)}$ denotes the value that the variable x takes on the kth iteration of the loop). Note that this table omits "uninteresting" recurrences (such as $q' = q + 0$) which indicate that a variable does not change in a loop. These closed forms are used to abstract the loop as follows:

Recurrence	Closed form
$r' = r - 1$	$r^{(k)} = r^{(0)} - k$
$t' = t - 1$	$t^{(k)} = t^{(0)} - k$

$$\varphi^{\circledast}_{\mathsf{inner}} = \boxed{\exists k.k \geq 0 \wedge r' = r - k \wedge t' = t - k \wedge \mathit{stable}(\{q, x, y\})}$$
$$= \boxed{r' = r + t' - t \wedge t' \leq t \wedge \mathit{stable}(\{q, x, y\})}$$

We may use this summary $\varphi^{\circledast}_{\mathsf{inner}}$ for the inner loop to compute a transition formula representing the body of the outer loop:

$$\varphi_{\mathsf{outer}} = [\![\langle v_2, v_3\rangle]\!] \odot [\![\langle v_3, v_4\rangle]\!] \odot \varphi^{\circledast}_{\mathsf{inner}} \odot [\![\langle v_4, v_7\rangle]\!] \odot [\![\langle v_7, v_2\rangle]\!]$$
$$= \boxed{q' = q + 1 \wedge r' = r + t' - y \wedge t' = 0 \wedge r \geq y \wedge \mathit{stable}(\{x, y\})}$$

We then apply the iteration operator to compute a transition formula for the outer loop. The recurrences found for the outer loop and their closed forms are shown to the right (again, with "uninteresting" recurrences omitted). We note that our algorithm extracts these recurrences from φ_{outer} using only semantic operations: the fact that φ_{outer} is an abstraction of a looping computation is completely transparent to the analysis. Using the closed forms of the recurrences to the right, we compute the following transition formula for the outer loop:

Recurrence	Closed form
$q' = q + 1$	$q^{(k)} = q^{(0)} + k$
$r' = r - y$	$r^{(k)} = r^{(0)} - y^{(0)}k$

$$\varphi^{\circledast}_{\mathsf{outer}} = \boxed{\exists k.k \geq 0 \wedge q' = q + k \wedge r' = r - ky \wedge \mathit{stable}(\{x, y\})}$$
$$= \boxed{q' \geq q \wedge r' = r - (q' - q)y \wedge \mathit{stable}(\{x, y\})}$$

Finally, we compute a transition formula which approximates all executions which end at v_8 as follows:

$$\varphi_P = [\![\langle v^{\mathsf{entry}}, v_1\rangle \cdot \langle v_1, v_2\rangle]\!] \odot \varphi^{\circledast}_{\mathsf{outer}} \odot [\![\langle v_2, v_8\rangle]\!]$$
$$= \boxed{q' \geq 0 \wedge r' = x - q'y \wedge r \leq y \wedge \mathit{stable}(\{x, y\})}$$

This formula is strong enough to imply that assertion $x' = q' * y' + r'$ holds at v_8. This is particularly interesting because it requires proving a *non-linear* transition invariant for the loop, which is out of scope for many state-of-the-art program analyzers.

3 Abstracting Loops with Linear Recurrence Analysis

In this section, we describe the iteration operator of linear recurrence analysis. Suppose that we have a formula φ_{body} which approximates the behaviour of the body of a loop. Our goal is to compute a formula $\varphi^{\circledast}_{\mathsf{body}}$ which represents the effect of zero or more executions of the loop body. Our iteration operator works by extracting recurrence relations from the formula φ_{body} and then computing closed forms for these relations. We present our iteration operator in three stages, based on the types of recurrence relations being considered: *simple recurrence equations*, *stratified recurrence equations*, and *linear recurrence (in)equations*. Simple and stratified recurrences are classical classes of recurrence equations. Linear recurrence (in)equations generalize the class of inequations presented in [2] by using stratified recurrences to generate polynomial (rather than just linear) inequations. The main conceptual contribution of this section is the idea to use SMT solvers to extract recurrences (and other relevant information) from a loop body formula.

In the remainder of this section, we fix a formula φ_{body} representing the body of a loop. We assume that φ_{body} is expressed in linear (rational and integer) arithmetic; our strategy for dealing with non-linear arithmetic is described in Section 4. We also assume that φ_{body} is satisfiable (if it is not, then we can take $\varphi^{\circledast}_{\mathsf{body}}$ to be $\bigwedge_{x \in \mathsf{Var}} x' = x$, which represents zero iterations of the loop).

3.1 Simple recurrence equations

We start by defining simple recurrences and induction variables.

Definition 1. *A simple recurrence for a formula φ is an equation of the form $x' = x + c$ (for a constant c) such that $\varphi \models x' = x + c$. If $x' = x + c$ is a simple recurrence for φ, we say that x* satisfies *the recurrence $x' = x + c$, and if there is some c such that x satisfies the recurrence $x' = x + c$, we say that x is an* induction variable.

Simple recurrences can be detected by first querying an SMT solver for a model m of φ_{body}, and then asking whether φ_{body} implies $x' = x + [\![x' - x]\!]^m$ (where $[\![x' - x]\!]^m$ denotes the interpretation of the term $x' - x$ in the model m). This implication holds iff x is an induction variable.

If x is an induction variable that satisfies the recurrence $x' = x + c$, then the closed form for x is $x^{(k)} = x^{(0)} + kc$ (writing $x^{(k)}$ for the value that x obtains on the kth iteration of the loop). To provide some early intuition on the iteration operator to be developed in the remainder of this section, let us suppose that

we are only interested in simple recurrences. Then a possible definition for the iteration operator is

$$\varphi^{\circledast}_{\mathsf{body}} \triangleq \boxed{\exists k \geq 0.\, \bigwedge\{x' = x + kc : x' = x + c \in SR(\varphi_{\mathsf{body}})\}}$$

where $SR(\varphi_{\mathsf{body}})$ is the set of simple recurrences satisfied by φ_{body}.

The iteration operator defined above is sound (it over-approximates the behaviour of any number of iterations of the loop, since each variable is either described exactly by a recurrence or is not constrained at all), but it is imprecise. The remainder of this section discusses more general recurrence equations which can be used to compute more precise transition invariants for loops.

3.2 Stratified recurrences equations

Consider the loop shown to the right. We can see that $\mathtt{x}$ satisfies a simple recurrence equation $\mathtt{x}' = \mathtt{x} + 1$, and that $\mathtt{y}$ satisfies a (non-simple) recurrence equation $\mathtt{y}' = \mathtt{y} + \mathtt{x} + 1$. A closed form for $\mathtt{y}$'s recurrence is $y^{(k)} = \mathtt{y}^{(0)} + \sum_{i=0}^{k-1}(\mathtt{x}^{(i)} + 1)$. Since $\mathtt{x}$ satisfies a simple recurrence ($x' = x + 1$), we have a closed form for $\mathtt{x}^{(i)}$, so we may simplify this recurrence and remove the summation:

```
while(x ≤ 10):
  x := x + 1
  y := y + x
  z := 2 * x
```

$$\mathtt{y}^{(k)} = \mathtt{y}^{(0)} + \sum_{i=0}^{k-1}(\mathtt{x}^{(0)} + i + 1) = \mathtt{y}^{(0)} + k\mathtt{x}^{(0)} + k + \sum_{i=0}^{k-1} i = \mathtt{y}^{(0)} + k\mathtt{x}^{(0)} + \frac{k(k+1)}{2}.$$

Stratified recurrence equations generalize this idea: starting from simple recurrence equations, we solve more and more complicated recurrences using the closed forms for simpler ones. As with the example above, stratified recurrences have non-linear closed forms. Non-linear invariant generation is not the main focus of our work, but it is sometimes a necessary intermediate step for proving *linear* invariants in a compositional setting: since our analysis cannot take advantage of contextual information when analyzing a loop, we generate a non-linear invariant and then, after the analysis has examined more context, simplify it (using the linearization algorithm from Section 4).

Definition 2. *Let φ be a formula. The stratified recurrence equations (and stratified induction variables) of φ are defined inductively as:*

- *A simple recurrence equation which is satisfied by φ is a stratified recurrence equation of φ (and a simple induction variable is a stratified induction variable) at stratum 0.*
- *Let $\boldsymbol{y}$ denote a vector of the stratified induction variables of strata $\leq N$. A recurrence of the form $x' = x + \boldsymbol{cy}$ (where $\boldsymbol{c}$ is a vector of constants) is a stratified recurrence at stratum $N + 1$ (and if x satisfies such a recurrence, it is a stratified induction variable at stratum $N + 1$).*

We use $\mathsf{siv}(\varphi)$ to denote the set of all stratified induction variables of φ.

Let us now discuss how stratified recurrences are detected from a loop body formula φ_{body}. We begin by computing the affine hull $\mathit{aff}(\varphi_{\mathsf{body}})$ of φ_{body} (Algo-

Algorithm 1: Affine hull.

Input : Satisfiable formula φ_{body}
Output: Affine hull of φ_{body}
$H \leftarrow \bot$; $\psi \leftarrow \varphi_{\mathsf{body}}$;
while *there exists a model* m *of* ψ **do**
  $H' \leftarrow \bigwedge\{x = [\![x]\!]^m : x \in \mathsf{Var} \cup \mathsf{Var}'\}$;
  $H \leftarrow H \sqcup^{=} H'$; `/*Join in the domain of linear equalities*/`
  $\psi \leftarrow \psi \wedge \neg H$;
end
return H

rithm 1).[1]

Definition 3. *The* affine hull *aff*(φ) *of a formula* φ *is the smallest affine set which contains* φ*, represented as (the set of solutions to) a system of equations* $A\boldsymbol{x} = \boldsymbol{b}$*, where* $\boldsymbol{x} = \left[x_1 \cdots x_n\ x'_1 \cdots x'_n\right]$*. Logically, aff*$(\varphi)$ *is a system of equations which satisfies the following three properties: (1)* $\varphi \models$ *aff*(φ)*, (2) every linear equation over* $\mathsf{Var} \cup \mathsf{Var}'$ *which is implied by* φ *is also implied by aff*(φ)*, and (3) no equation in aff*(φ) *is implied by the others.*

Our strategy for detecting stratified recurrences is based on the following lemma. Combined with property (2) of *aff*$(\varphi_{\mathsf{body}})$ above, this lemma implies that any equation implied by φ_{body} can be expressed as a linear combination of the equations in *aff*$(\varphi_{\mathsf{body}})$.

Lemma 1 ([28], Corollary 3.1d). *Let* A *be a matrix,* $\boldsymbol{b}$ *be a column vector,* $\boldsymbol{c}$ *be a row vector, and* d *be a constant. Assume that the system* $A\boldsymbol{x} = b$ *has a solution. Then* $A\boldsymbol{x} = \boldsymbol{b}$ *implies* $\boldsymbol{c}\boldsymbol{x} = d$ *iff there is a row vector* $\boldsymbol{\lambda}$ *such that* $\boldsymbol{\lambda}A = \boldsymbol{c}$ *and* $\boldsymbol{\lambda}\boldsymbol{b} = d$*.*

Let us write *aff*$(\varphi_{\mathsf{body}})$ as $A\boldsymbol{x} = \boldsymbol{b}$. Suppose that we have detected all recurrences of strata $< N$, and that we want to determine whether a variable x_i $(0 \leq i \leq n)$ is an induction variable at stratum N. Then we ask whether there exists $\boldsymbol{\lambda}$, $\boldsymbol{c}$, and d such that:

- $\boldsymbol{\lambda}A = \boldsymbol{c}$ and $\boldsymbol{\lambda}\boldsymbol{b} = d$ (i.e, $\boldsymbol{c}\boldsymbol{x} = d$ is implied by *aff*$(\varphi_{\mathsf{body}})$ and thus by φ_{body})
- $c_i = 1$ and $c_{i+n} = -1$ (the coefficients of x_i and x'_i are 1 and -1, respectively)
- For all j such that $j \neq i + n$ and $n \leq j \leq 2n$, $c_j = 0$ (except for x'_i, all coefficients of primed variables are 0).
- For all j such that $j \neq i$ such that x_j is not an induction variable of strata $< N$ and $n \leq j \leq 2n$, $c_j = 0$ (except for x_i and induction variables of strata $< N$, all coefficients for unprimed variables are 0).

Thus, after computing the affine hull of φ_{body}, determining whether a given variable satisfies a stratified recurrence is simply a matter of solving a system of linear equations (e.g., using Gaussian elimination).

[1] This algorithm is a specialization of the one in [26] to the abstract domain of linear equalities.

Closed forms for stratified recurrences. We first state a lemma:

Lemma 2. *The closed form for a stratified induction variable of strata N is of the form*

$$x^{(k)} = p_0(k) + p_1(k)y_1^{(0)} + \cdots + p_n(k)y_n^{(0)}$$

where each y_i is a stratified induction variable of strata $< N$ and each $p_i(k) \in \mathbb{Q}[k]$ is a polynomial of one variable with rational coefficients.

Our algorithm for solving stratified recurrences is based on a constructive proof for this lemma. We proceed by induction on strata. The base case is trivial. Suppose that we have a recurrence at strata N (and all $y_1, ..., y_n$ are of strata $< N$): $x' = x + c_1y_1 + \cdots + c_ny_n + b$. Then we may write $x^{(k)} = x^{(0)} + \sum_{i=0}^{k-1} \left(c_1y_1^{(i)} + \cdots + c_ny_n^{(i)} + b\right)$. By our induction hypothesis, each $y_j^{(i)}$ can be written as a linear term with coefficients from $\mathbb{Q}[k]$. It follows that there exists $p_0, ..., p_n \in \mathbb{Q}[k]$ so that

$$c_1y_1^{(i)} + \cdots + c_ny_n^{(i)} + b = p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)}$$

Thus we have

$$\begin{aligned} x^{(k)} &= x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)} \\ &= x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + y_1^{(0)} \sum_{i=0}^{k-1} p_1(i) + \cdots + y_n^{(0)} \sum_{i=0}^{k-1} p_n(i) \end{aligned}$$

The closed form of a summation of a polynomial of degree m is a polynomial of degree $m+1$. We can find this polynomial via curve fitting (i.e., we compute the first $m+1$ terms of the summation and then solve the corresponding linear system of equations for the coefficients of the polynomial).

3.3 Linear recurrence (in)equations

```
while(x ≥ 0 ∧ y ≥ 0):
  if(*): x := x - 1
  else: y := y - 1
```

Recurrence equations (such as the simple and stratified varieties) yield very accurate approximations for *some* variables, but what about variables which do not satisfy *any* recurrence equation? For example, consider that neither $\mathtt{x}$ nor $\mathtt{y}$ satisfy a recurrence equation in the loop to the right. However, they *do* satisfy recurrence *inequations*: $\mathtt{x} - 1 \leq \mathtt{x}'$, $\mathtt{x}' \leq \mathtt{x}$, $\mathtt{y} - 1 \leq \mathtt{y}'$, and $\mathtt{y}' \leq \mathtt{y}$. These inequations can be closed to yield $\mathtt{x}^{(0)} - k \leq \mathtt{x}^{(k)}$ and $\mathtt{x}^{(k)} \leq \mathtt{x}^{(0)}$, $y^{(0)} - k \leq y^{(k)}$, and $y^{(k)} \leq y^{(0)}$. In this section, we discuss linear recurrence (in)equations, which allow us to compute good approximations for loops that cannot be completely described by recurrence equations.

Definition 4. *A* linear recurrence (in)equation *of a formula φ is an (in)equation which is implied by φ and which is of the form*

$$\boldsymbol{c}\boldsymbol{x}' \bowtie \boldsymbol{c}\boldsymbol{x} + \boldsymbol{b}\boldsymbol{y} + d$$

where $\bowtie \in \{<, \leq, =\}$, $\boldsymbol{x}$ is any vector of variables, $\boldsymbol{y}$ is a vector of stratified induction variables in φ_{body}, $\boldsymbol{c}$, $\boldsymbol{b}$ are constant vectors, and d is a constant.

Linear recurrence (in)equations generalize recurrence equations in two ways: first, they allow for *inequalities* rather than equations. Second, they allow recurrences for *linear terms*, rather than just variables. For example, the linear recurrence equation $(\mathtt{x}' + \mathtt{y}') = (\mathtt{x} + \mathtt{y}) + 1$ is satisfied by the body of the loop above, which can be closed to yield $(\mathtt{x}^{(k)} + \mathtt{y}^{(k)}) = (\mathtt{x}^{(0)} + \mathtt{y}^{(0)}) + k$.

We now describe our method for detecting and solving linear recurrence (in)equations. We begin by introducing a set of *difference variables* δ_x, one for each variable $x \notin \mathsf{siv}(\varphi_{\mathsf{body}})$ (variables which do belong to $\mathsf{siv}(\varphi_{\mathsf{body}})$ are already precisely described by recurrence equations, so we need not approximate them). We then compute (via Algorithm 2) the convex hull of the formula ψ defined as:

$$\psi \triangleq \exists X.\varphi_{\mathsf{body}} \wedge \bigwedge\{\delta_x = x' - x : x \in \mathsf{Var} \setminus \mathsf{siv}(\varphi_{\mathsf{body}})\}$$

where X is $\mathsf{Var}' \cup (\mathsf{Var} \setminus \mathsf{siv}(\varphi_{\mathsf{body}}))$.

Algorithm 2: Convex hull.

Input : Satisfiable formula ψ, set of variables X
Output: Convex hull of $\exists X.\psi$
$P \leftarrow \bot$;
while *there exists a model* m *of* ψ **do**
  Let Q be a cube of the DNF of ψ s.t. $m \models Q$;
  $Q \leftarrow \mathit{project}(Q, X)$; `/*Polyhedral projection*/`
  $P \leftarrow P \sqcup Q$; `/*Polyhedral join*/`
  $\psi \leftarrow \psi \wedge \neg P$;
end
return P

Geometrically, the convex hull $\mathit{hull}(\varphi_{\mathsf{body}})$ is the smallest convex polyhedron which contains φ_{body}. Logically, it is a set of (in)equations such that (1) every (in)equation in $\mathit{hull}(\varphi_{\mathsf{body}})$ is implied by φ_{body}, and (2) any linear (in)equation (over $\mathsf{Var} \cup \mathsf{Var}'$) which is implied by φ_{body} is also implied by $\mathit{hull}(\varphi_{\mathsf{body}})$. For example, $\mathit{hull}(\varphi_{\mathsf{body}})$ for the loop above is:

$$0 \leq \delta_x \wedge \delta_x \leq 1 \wedge 0 \leq \delta_y \wedge \delta_y \leq 1 \wedge \delta_x + \delta_y = 1$$

We note that the only variables which appear in the (in)equations in $\mathit{hull}(\varphi_{\mathsf{body}})$ are (stratified) induction variables and difference variables. Thus, we may write any (in)equation in $\mathit{hull}(\varphi_{\mathsf{body}})$ as $\boldsymbol{c\delta} \bowtie \boldsymbol{by} + d$ (where $\boldsymbol{\delta}$ is the vector of difference variables, $\boldsymbol{y}$ is the vector of stratified induction variables, $\boldsymbol{c}$ and $\boldsymbol{b}$ are constant vectors, and d is a constant). Recalling the definition of the difference variables, we may rewrite such an inequation as $\boldsymbol{c(x' - x)} \bowtie \boldsymbol{by} + d$ and then rewrite again as $\boldsymbol{cx'} \bowtie \boldsymbol{cx} + \boldsymbol{by} + d$, which matches the definition of linear recurrence (in)equations given in Definition 4.

We may close such a linear recurrence (in)equation as follows:

$$\boldsymbol{cx}^{(k)} \bowtie \boldsymbol{cx}^{(0)} + \sum_{i=0}^{k-1} \boldsymbol{by}^{(i)} + d$$

We can compute a closed form for the summation $\sum_{i=0}^{k-1} \boldsymbol{by}^{(i)} + d$ as in the preceding section.

3.4 Loop guards

A loop body typically contains crucial information about the execution of the loop that cannot be captured by recurrence relations. For example, consider the loop in Section 3.2. Supposing that the loop executes n times, we must have that $\mathtt{x}^{(k)} \leq 10$ for each $k < n$. Further, consider that the variable $\mathtt{z}$ is a function of the simple induction variable $\mathtt{x}$, and so $\mathtt{z}^{(k)}$ can be described precisely in terms of the pre-state variables (even though it does not itself satisfy any recurrence):

$$\mathtt{z}^{(k)} = \begin{cases} \mathtt{z}^{(0)} & \text{if } k = 0 \\ 2(\mathtt{x}^{(0)} + k + 1) & \text{otherwise.} \end{cases}$$

The question is: how can we recover this type of information from a loop body formula?

We define the *guard* of a transition formula φ as follows:

$$guard(\varphi) \triangleq \boxed{(\exists \mathsf{Var}.\varphi) \wedge (\exists \mathsf{Var}'.\varphi)}$$

If φ is a loop body formula, then $guard(\varphi)$ is a formula which over-approximates the effect of executing *at least one* execution of the loop. Intuitively, $(\exists \mathsf{Var}.\varphi)$ as a precondition that must hold before every iteration of the loop and $(\exists \mathsf{Var}'.\varphi)$ as a post-condition of the loop that must hold after each iteration.

Consider again the example loop in Section 3.2, we have the following loop body formula

$$\varphi_{\mathsf{body}} = \boxed{x \leq 10 \wedge x' = x + 1 \wedge y' = y + x' \wedge z' = 2x'}$$

We compute $guard(\varphi_{\mathsf{body}})$ as follows:

$$\begin{aligned} guard(\varphi_{\mathsf{body}}) &= \boxed{(\exists \mathtt{x}, \mathtt{y}, \mathtt{z}.\varphi_{\mathsf{body}}) \wedge (\exists \mathtt{x'}, \mathtt{y'}, \mathtt{z'}.\varphi_{\mathsf{body}})} \\ &\equiv \boxed{(\mathtt{x} \leq 10) \wedge (\mathtt{x'} \leq 11 \wedge \mathtt{z} = 2\mathtt{x'})}, \end{aligned}$$

and thereby recover the desired information about $\mathtt{x}$ and $\mathtt{z}$.

Since loop body formulas may be large, it may be adventageous in practice to simplify the guard formula by eliminating the quantifiers (as we did above). A second option, which is more efficient but less precise, is to over-approximate quantifier elimination. Two possibilities are to use Algorithm 2 to compute the convex hull of $guard(\varphi_{\mathsf{body}})$, or to use optimization modulo theories [19] to compute intervals for each pre- and post-state variable in φ_{body}.

3.5 Bringing it all together

We close this section by describing how the pieces defined in this section fit into the iteration operator of linear recurrence analysis. We let $CR(\varphi_{\mathsf{body}})$ denote the set of closed linear recurrence (in)equations (including simple and stratified recurrence equations) satisfied by φ_{body}. Each such (in)equation is of the form

$\boldsymbol{cx}^{(k)} \bowtie t$, where the free variables of t are drawn from $\{x^{(0)} : x \in \mathsf{Var}\}$ and a distinguished variable $k \notin \mathsf{Var}$ indicating the loop iteration. We define

$$\varphi^{+}_{\mathsf{body}} \triangleq \boxed{\exists k.k \geq 1 \wedge \bigwedge\{\boldsymbol{cx'} \bowtie t[\boldsymbol{x}^{(0)} \mapsto \boldsymbol{x}] : \boldsymbol{cx'} \bowtie t \in CR(\varphi_{\mathsf{body}})\}}$$

where $t[\boldsymbol{x}^{(0)} \mapsto \boldsymbol{x}]$ denotes the term t with every variable of the form $x^{(0)}$ is replaced by the corresponding variable x.

Finally, our iteration operator is defined as:

$$\varphi^{\circledast}_{\mathsf{body}} \triangleq \boxed{(\varphi^{+}_{\mathsf{body}} \wedge \mathit{guard}(\varphi_{\mathsf{body}})) \vee \textstyle\bigwedge_{x \in \mathsf{Var}} x' = x.}$$

4 Linearization

The iteration operator presented in the previous section relies heavily on using an SMT solver to extract information from loop body formulas. This strategy requires that loop body formulas are expressed in a decidable theory which is supported by SMT solvers (in particular, linear arithmetic). However, a program may contain non-linear instructions, and even if it does not, our iteration operator may introduce non-linearity (consider Example 1, where the transition formula for the outer loop $\varphi^{\circledast}_{\mathsf{outer}}$ contains the non-linear proposition $r' = x - q'y$). Our solution to this problem is to *linearize* non-linear formulas before passing them to the iteration operator.

Linearization is an operation that, given an (arbitrary) arithmetic formula φ, computes a formula $\mathit{lin}(\varphi)$ which over-approximates φ (i.e., $\varphi \Rightarrow \mathit{lin}(\varphi)$), but which is expressed in linear arithmetic. There is generally no best approximation of a non-linear formula as a linear formula, so our method is (necessarily) a heuristic.

We explain our linearization algorithm informally using an example. Consider the following non-linear formula (where w, x, y, z are integers):

$$\psi \triangleq 1 \leq w = x < y < 5 \wedge w * y \leq z \leq x * y$$

Our algorithm begins by normalizing ψ, separating it into a linear part and a set of non-linear equations (introducing existentially quantified temporary variables as necessary). For example, the result of normalizing ψ is:

$$\big(1 \leq w = x < y < 5 \wedge \leq \gamma_0 \leq z \leq \gamma_1\big) \wedge \big(\gamma_0 = w * y \wedge \gamma_1 = x * y\big)$$

The left conjunct is a linear over-approximation of ψ, but it is very imprecise: semantically equal (but syntactically distinct) non-linear terms become semantically *un*equal in the over-approximation, and all information about the magnitude of non-linear terms is lost. To increase precision of this approximation, we use two strengthening steps.

1. We replace the non-linear operations with uninterpreted function symbols and then compute the affine hull of the resulting formula to infer equalities between non-linear terms. For our example ψ, the we discover that $\gamma_0 = \gamma_1$.
2. We compute concrete and symbolic intervals for non-linear terms. Consider $\gamma_0 = x * y$ from our example ψ. We first compute concrete ($x \in [1, 3]$ and $y \in [2, 4]$) and symbolic ($x \in [x, x]$ and $y \in [y, y]$) intervals for the operands

x and y, using symbolic optimization [19] to compute the concrete intervals. We obtain a concrete interval for $x * y$ ($x * y \in [2, 12]$) by multiplying the concrete intervals of its operands. We obtain symbolic intervals for $x * y$ ($x * y \in [y, 3y]$ and $x * y \in [2x, 4x]$) by multiplying the concrete interval for x by the symbolic interval for y and vice-versa. As a result of interval computation, we discover: $2 \leq \gamma_1 \leq 12 \wedge y \leq \gamma_1 \leq 3y \wedge 2x \leq \gamma_1 \leq 4x$

Finally, we take $lin(\psi)$ to be the initial coarse linear approximation of ψ conjoined with the facts discovered by the two strengthening steps.

We expect linearization to have broad applications outside of the context in which we presented it, particularly in program analysis, where over-approximation can be tolerated but non-linear terms cannot. Finding improved linearization heuristics is an interesting direction of future work.

5 Experiments

We wrote a tool which implements LRA and analyzes C code (using the CIL [24] frontend).[2] We use Z3 [9] to resolve SMT queries that result from applying the iteration operator and checking assertion violations. Polyhedra operations are passed to the New Polka library implemented in Apron [4]. The quantifier elimination algorithm from [22] is used to compute loop guards.

We tested two different configurations of LRA: one which is fully compositional (LRA-Comp) and does not take advantage of contextual information, and one (LRA) which uses an intraprocedural polyhedron analysis [8] to gain *some* contextual information, but which is otherwise compositional. We compare LRA's performance against the state-of-the-art invariant generation and verification tools CPAchecker (overall winner of the 2015 Software Verification Competition) and SeaHorn (winner of the loops category among tools which are sound for verification).

To evaluate the precision of LRA we used it to verify the correctness of a suite of 119 small loop benchmarks of varying difficulty. Our benchmark suite was drawn from the *loops* category of the 2015 Software Verification Competition (SVComp-15), as well as a set of *non-linear* benchmarks (Non-linear), such as the one in Figure 1. The results for the 81 safe, integer-only benchmarks from these suites are shown in Table 1. The suite also contains 38 *unsafe* benchmarks: LRA and LRA-Comp have no false negatives on these benchmarks; CPAchecker has 3 and SeaHorn has 2.

Our results demonstrate that LRA is an effective invariant generation algorithm. Even the fully compositional variant of LRA (LRA-Comp) is able to prove safety for 80% of the benchmarks we considered). We also note that there are 8 benchmarks for which LRA can prove safety but which CPAchecker and SeaHorn cannot.

[2] The tool and benchmarks are available at `http://cs.toronto.edu/~zkincaid/lra`.

Benchmark suite	# Bench	LRA	LRA-Comp	CPAChecker	SeaHorn
SVComp-15	74	65	60	37	65
Non-linear	7	6	5	1	3
Total	81	71 (88%)	65 (80%)	38 (47%)	68 (85%)
Running time across all benchmark suites					
Mean		5.4s	3.0s	42.4s	37.7s
Median		0.8s	0.8s	1.6s	0.2s

Table 1. Experimental results.

6 Related work

There is a great deal of work on compositional invariant generation and acceleration which is related to the technique described in this paper. In this section, we compare our technique to a sampling of this work.

Recurrence analysis. The idea of using closed forms of recurrence relations to approximate loops has appeared in a number of other papers. Generally speaking, our work differs from previous work in two essential ways: first, we use an SMT solver to extract *semantic* recurrences, rather than *syntactic* recurrences. Second, we consider *approximate recurrences* (inequations over linear terms) rather than exact recurrences (equations over variables). A survey of some of this work follows.

Ammarguellat and Harrison present a method for detecting induction variables which is compositional in the sense that it uses closed forms for inner loops in order to recognize nested recurrences [1]. Maps from variables to symbolic terms (effectively a symbolic constant propagation domain) is used as the abstract domain. Kovács presents a technique for discovering invariant polynomial equations based on solving recurrence relations [15]. The simple and stratified recurrence equations considered in this paper are a strict subset of the recurrences considered in [15], but our algorithm for solving recurrences is simpler. Kroening et al. [16] presents a technique for computing *under*-approximations of loops which uses polynomial curve-fitting to directly compute closed forms for recurrences rather than extracting recurrences and then solving them in a separate step.

Ancourt et al. present a method for computing recurrence inequations for while loops with affine bodies [2]. Like the method we present on Section 3.3, their method is based on using difference variables and polyhedral projections. Our method generalizes this work by (1) extending it to arbitary control flow, with (possibly non-linear) formulas as bodies rather than affine transformations, (2) integrating recurrence inequations with stratified induction variables, thereby allowing enabling the computation of invariant polynomial inequations. Ancourt et al. briefly discuss a method for computing invariant polynomial inequations, but it is based on higher-order differences rather than stratified recurrence inequations. For example, in Figure 1, the analysis discussed in [2] would be able to prove that `r` is decremented by a constant amount at every loop iteration, but could not prove that the constant amount is exactly `y`.

Acceleration. *Acceleration* is a technique closely related to recurrence analysis that was pioneered in infinite-state model checking [6,11,3], and which has recently found use in program analysis [12,18,13]. Given a set of reachable states and an affine transformation describing the body of a loop, acceleration computes an *exact* post-image which describes the set of reachable states after executing any number of iterations of the loop (although there is recent work on *abstract acceleration* uses computes over-approximate post-images [12,13]). In contrast, our technique is *approximate* rather than exact, and computes loop summaries rather than post-images. A result of these two features is that our analysis to be applied to arbitrary loops, while acceleration is classically limited to simple loops where the body consists of a sequence of assignment statements.

Compositional program analysis. Compositional program analysis has a long history. Particular examples are interprocedural analyses based on summarization [29] and elimination-style dataflow analyses (a good overview of which can be found in [27]). The following surveys recent work on compositional analysis for numerical invariants.

Kroening et al. [17] and Biallas et al. [5] present compositional analysis techniques based on predicate abstraction. In addition to predicate abstraction, there are a few papers which use numerical abstract domains for compositional analysis. These include an algorithm for detecting affine equalities between program variables [23], an algorithm for detecting polynomial equalities between program variables [7], a disjunctive polyhedra analysis which uses widening to compute loop summaries [25], and a method for automatically synthesizing transfer functions for template abstract domains using quantifier elimination [21]. Our abstract domain is the set of arbitrary arithmetic formula, which is more expressive than these domains, but which (as usual) incurs a price in performance. It would be interesting to apply abstractions to our formulas to improve the performance of our analysis.

Linearization. Our linearization algorithm was inspired by Miné's procedure for approximating non-linear abstract transformers [20]. Miné's procedure abstracts non-linear terms by linear terms with interval coefficients using the abstract value in the pre-state to derive intervals for variables. Our algorithm abstracts non-linear terms by sets of symbolic and concrete intervals, and applies to the more general setting of approximating arbitrary formulas.

7 Conclusion

This paper presents a fully compositional algorithm for generating numerical invariants of imperative programs. Our method for abstracting loops makes essential use of compositionality: we assume that we are given a formula which approximates the body of a loop, and we use an SMT solver to extract recurrence relations and then use the closed forms of these recurrences to approximate the loop. We have demonstrated experimentally that our method is competitive with leading invariant generation and verification tools.

References

1. Z. Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. PLDI, pages 283–295, 1990.
2. C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1):3–16, Oct. 2010.
3. S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA*, pages 474–488. 2005.
4. J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.
5. S. Biallas, J. Brauer, A. King, and S. Kowalewski. Loop leaping with closures. In *SAS*, pages 214–230, 2012.
6. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV*, pages 55–67. 1994.
7. M. A. Colón. Approximating the algebraic relational semantics of imperative programs. In *SAS*, pages 296–311. 2004.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.
9. L. De Moura and N. Bjørner. Z3: an efficient SMT solver. TACAS, pages 337–340, 2008.
10. A. Farzan and Z. Kincaid. An algebraic framework for compositional program analysis. *CoRR*, abs/1310.3481, 2013.
11. A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST TCS*, pages 145–156, 2002.
12. L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS*, pages 144–160. 2006.
13. B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. In *POPL*, pages 529–540, 2014.
14. S. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. Mccarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.
15. L. Kovács. Reasoning algebraically about P-solvable loops. In *TACAS*, pages 249–264. 2008.
16. D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *CAV*, pages 381–396. 2013.
17. D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger. Loop summarization using abstract transformers. In *ATVA*, pages 111–125. 2008.
18. J. Leroux and G. Sutre. Accelerated data-flow analysis. In *SAS*, pages 184–199, 2007.
19. Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *POPL*, pages 607–618, 2014.
20. A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.
21. D. Monniaux. Automatic modular abstractions for linear constraints. In *POPL*, pages 140–151, 2009.
22. D. Monniaux. Quantifier elimination by lazy model enumeration. In *CAV*, pages 585–599, 2010.

23. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. POPL, pages 330–341, 2004.
24. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
25. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. ASIAN, pages 331–345, 2007.
26. T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.
27. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, Sept. 1986.
28. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.
29. M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
30. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.